(1)

UMIACS-TR-90-40
CS-TR-2437

March 1990

Recursive Star-Tree Parallel Data-Structure

Omer Berkman and *Uzi Vishkin*†
Institute for Advanced Computer Studies and
†Department of Electrical Engineering
University of Maryland
College Park, MD 20742
and
Tel Aviv University

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS–TR–90–40
CS–TR–2437

March 1990

Recursive Star-Tree Parallel Data-Structure

Omer Berkman and *Uzi Vishkin*†
Institute for Advanced Computer Studies and
†Department of Electrical Engineering
University of Maryland
College Park, MD 20742
and
Tel Aviv University

ABSTRACT

This paper introduces a novel parallel data-structure, called recursive STAR-tree (denoted '* tree'). For its definition, we use a generalization of the * functional [1]. Using recursion in the spirit of the inverse-Ackermann function, we derive recursive *-trees.

The recursive *-tree data-structure leads to a new design paradigm for parallel algorithms. This paradigm allows for:

- Extremely fast parallel computations. Specifically, $O(\alpha(n))$ time (where $\alpha(n)$ is the inverse of Ackermann function) using an optimal number of processors on the (weakest) CRCW PRAM.
- These computations need only constant time, using an optimal number of processors if the following non-standard assumption about the model of parallel computation is added to the CRCW PRAM: an extremely small number of processors can write simultaneously each into different bits of the same word.

Applications include:

(1) A new algorithm for finding lowest common ancestors in trees which is considerably simpler than the known algorithms for the problem.
(2) Restricted domain merging.
(3) Parentheses matching.
(4) A new parallel reducibility.

† The research of this author was supported by NSF grants CCR-8615337 and CCR-8906949 and ONR grant N00014-85-K-0046.

[1] Given a real function f, denote $f^{(1)}(n) = f(n)$ and $f^{(i)}(n) = f(f^{(i-1)}(n))$ for $i \geq 1$. The * functional maps f into another function $*f$. $*f(n) = \textit{minimum}\{i \mid f^{(i)}(n) \leq 1\}$. If this minimum does not exist then $*f(n) = \infty$.

1. Introduction

The model of parallel computation that is used in this paper is the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). We assume that several processors may attempt to write at the same memory location only if they are seeking to write the same value (the so called, Common CRCW PRAM). We use the weakest Common CRCW PRAM model, in which only concurrent writes of the value one are allowed. Given two parallel algorithms for the same problem one is *more efficient* than the other if: (1) primarily, its time-processor product is smaller, and (2) secondarily (but important), its parallel time is smaller. *Optimal* parallel algorithms are those with a linear time-processor product. A *fully-parallel* algorithm is a parallel algorithm that runs in constant time using an optimal number of processors. An *almost fully-parallel* algorithm is a parallel algorithm that runs in $\alpha(n)$ (the inverse of Ackermann function) time using an optimal number of processors.

The notion of fully-parallel algorithm represents an ultimate theoretical goal for designers of parallel algorithms. Research on lower bounds for parallel computation (see references later) indicates that for nearly any interesting problem this goal is unachievable. These same results also preclude almost fully-parallel algorithms for the same problems. Therefore, any result that approaches this goal is somewhat surprising.

The class of doubly logarithmic optimal parallel algorithms and the challenge of designing such algorithms is discussed in [BBGSV-89]. The class of almost fully-parallel algorithms represents an even more strict demand.

There is a remarkably small number of problems for which there exist optimal parallel algorithms that run in $o(\text{loglog}n)$ time. These problems include: (a) OR and AND of n bits. (b) Finding the minimum among n elements, where the input consists of integers in the domain $[1,...,n^c]$ for a constant c. See Fich, Ragde and Wigderson [FRW-84]. (c) $\log^{(k)}n$-coloring of a cycle where $\log^{(k)}$ is the k'th iterate of the log function and k is constant [CV-86a]. (d) Some probabilistic computational geometry problems, [S-88]. (e) Matching a pattern string in a text string, following a processing stage in which a table based on the pattern is built [Vi-89].

Not only that the number of such upper bounds is small, there is evidence that for almost any interesting problem an $o(\text{loglog}n)$ time optimal upper bound is impossible. We mention time lower bounds for a few very simple problems. Clearly, these lower bounds apply to more involved problems. For brevity, only lower bounds for optimal speed up algorithms are stated. (a) Parity of n bits. The time lower bound is $\Omega(\log n/\text{loglog}n)$. This follows from the lower bound of [H-86] for circuits together with the general simulation result of [StV-84], or from Beame and Hastad [BHa-87]. (b) Finding the minimum among n elements. Lower bound: $\Omega(\text{loglog}n)$ on comparison model, [Va-75]. Same lower bound for (c) Merging two sorted arrays of numbers, [BHo-85].

The main contribution of this paper is a parallel data-structure, called *recursive *-tree*. This data-structure provides also a new paradigm for parallel algorithms. There are two known examples where tree based data-structures provide a "skeleton" for parallel algorithms:
(1) Balanced binary trees. The depth of a balanced binary tree with n leaves is $\log n$.
(2) "Doubly logarithmic" balanced trees. The depth of such a tree with n leaves is $\log\log n$. Each node of a doubly logarithmic tree, whose rooted subtree has x leaves, has $\sqrt{x}$ children.
Balanced binary trees are used in the prefix-sums algorithm of [LF-80] (that is perhaps the most heavily used routine in parallel computation) and in many other logarithmic time algorithms. [BBGSV-89] show how to apply doubly logarithmic trees for guiding the flow of the computation in several doubly logarithmic algorithms (including some previously known algorithms). Similarly, the recursive *-tree data-structure provides a new pattern for almost fully-parallel algorithms.

In order to be able to list results obtained by application of *-trees we define the following family of extremely slow growing functions. Our definition is direct. A subsequent comment explains how this definition leads to an alternative definition of the inverse-Ackermann function. For a more standard definition see [Ta-75]. We remark that such direct definition is implicit in several "inverse Ackermann related" serial algorithms (e.g., [HS-86]).

The Inverse-Ackermann function

Consider a real function f. Let $f^{(i)}$ denote the i-th iterate of f. (Formally, we denote $f^{(1)}(n) = f(n)$ and $f^{(i)}(n) = f(f^{(i-1)}(n))$ for $i \geq 1$.) Next, we define the * (pronounced "star") functional that maps the function f into another function $^*f$. $^*f(n) = \textit{minimum}\{i \mid f^{(i)}(n) \leq 1\}$. (*Notational comment.* Note that the function $\log^*$ will be denoted $^*\log$ using our notation. This change is for notational convenience.)

We define inductively a series I_k of slow growing functions:
(i) $I_0(n) = n-2$, and (ii) $I_k = {}^*I_{k-1}$.

The first four in this series are familiar functions: $I_0(n) = n-2$, $I_1(n) = \lfloor n/2 \rfloor$, $I_2(n) = \lfloor \log n \rfloor$ and $I_3(n) = \lfloor {}^*\log n \rfloor$.

The "inverse-Ackermann" function is $\alpha(n) = \textit{minimum}\ \{i \mid I_i(n) < i\}$. See the following comment.

Comment. Ackermann's function is defined as follows:
$A(0,0) = 0$; $A(i,0) = 1$, for $i > 0$; $A(0,j) = j+2$, for $j > 0$; and
$A(i,j) = A(i-1, A(i,j-1))$, for $i,j > 0$.

It is interesting to note that I_k is actually the inverse of the k'th recursion level of A, the Ackermann function. Namely: $I_k(n) = \textit{minimum}\ \{i \mid A(k,i) \geq n\}$ or $I_k(A(k,n)) = n$. The definition of $\alpha(n)$ is equivalent to the more often used definition (but perhaps less intuitive): $\textit{minimum}\ \{i \mid A(i,i) \geq n\}$.

Applications of recursive *-trees

1. *The lowest-common-ancestor (LCA) problem.* Suppose a rooted tree T is given for preprocessing. The preprocessing should enable a single processor to process quickly queries of the following form. Given two vertices u and v, find their lowest common ancestor in T.

Results. (i) Preprocessing in $I_m(n)$ time using an optimal number of processors. Queries will be processed in $O(m)$ time that is $O(1)$ time for constant m. A more specific result is: (ii) almost fully-parallel preprocessing and $O(\alpha(n))$ for processing a query. These results assume that the Euler tour of the tree and the level of each vertex in the tree are given. Without this assumption the time for preprocessing is $O(\log n)$, using an optimal number of processors, and each query can be processed in constant time. For a serial implementation the preprocessing time is linear and a query can be processed in constant time.

Significance: Our algorithm for the LCA problem is new and is based on a completely different approach than the serial algorithm of Harel and Tarjan [HT-84] and the simplified and parallelizable algorithm of Schieber and Vishkin [ScV-88]. Its serial version is considerably simpler than these two algorithms. Specifically, consider the Euler tour of the tree and replace each vertex in the tour by its level. This gives a sequence of integers. Unlike previous approaches the new LCA algorithm is based only on analysis of this sequence of integers. This provides another interesting example where the quest for parallel algorithms enriches also the field of serial algorithms. Algorithms for quite a few problems use an LCA algorithm as a subroutine. We mention some: (1) Strong orientation [Vi-85]; (2) Computing open ear-decomposition and st-numbering of a biconnected graph [MSV-86]; also [FRT-89] and [RR-89] use as a subroutine an algorithm for st-numbering and thus also the LCA algorithm. (3) Approximate string matching on strings [LV-88] and on trees [SZ-89], and retrieving information on strings from their suffix trees [AILSV-88].

2. *The all nearest zero bit problem.* Let $A=(a_1,a_2,\cdots,a_n)$ be an array of bits. Find for each bit a_i the nearest zero bit both to its left and right.

Result: An almost fully-parallel algorithm.

Literature. A similar problem was considered by [CFL-83] where the motivation was circuits.

3. *The parentheses matching problem.* Suppose we are given a legal sequence of parentheses. Find for each parenthesis its mate.

Result. Assuming the level of nesting of each parenthesis is given, we have an almost fully-parallel algorithm. Without this assumption $T=O(\log n/\log\log n)$, using an optimal number of processors.

Literature. Parentheses matching in parallel was considered by [AMW-88], [BSV-88], [BV-85] and [DS-83].

Remark. The algorithm for the parentheses matching is delayed into a later paper ([BeV-90a]), in order to keep this paper within reasonable length.

4. *Restricted domain merging.* Let $A=(a_1,...,a_n)$ and $B=(b_1,...,b_n)$, be two non-decreasing lists, whose elements are integers drawn from the domain $[1,...,n]$. The problem is to merge them into a sorted list.

Result: An almost fully-parallel algorithm.

Literature. Merging in parallel was considered by [Van-89], [BHo-85], [Kr-83], [SV-81] and [Va-75].

Remark. The merging algorithm is presented in another paper ([BeV-90]). The length of this paper was again a concern. Also, we recently found a way to implement it on a less powerful model (CREW PRAM) with the same bounds, and somewhat relax the restricted-domain limitation using unrelated techniques.

5. *Almost fully-parallel reducibility.* Let A and B be two problems. Suppose that any input of size n for problem A can be mapped into an input of size $O(n)$ for problem B. Such mapping from A to B is an *almost fully-parallel reducibility* if it can be realized by an almost fully-parallel algorithm.

Given a convex polygon, the *all nearest neighbors (ANN)* problem is to find for each vertex of the polygon its nearest (Euclidean) neighbor. Using almost fully-parallel reducibilities we prove the following lower bound for the ANN problem: Any CRCW PRAM algorithm for the ANN problem that uses $O(n\log^c n)$ (for any constant c) processors needs $\Omega(\log\log n)$ time.

We note that this lower bound was proved in [ScV-88a] using a considerably more involved technique.

Fully-parallel results

For our fully-parallel results we introduce the CRCW-bit PRAM model of computation. In addition to the above definition of the CRCW PRAM we assume that a few processors can write simultaneously each into different bits of the same word. Specifically, in our algorithms this number of processors is very small and never exceeds $O(I_d(n))$, where d is a constant. Therefore, the assumption looks to us quite reasonable from the architectural point of view. We believe that the cost for implementing a step of a PRAM on a feasible machine is likely to absorb implementation of this assumption at no extra cost. Though, we do not suggest to consider the CRCW-bit PRAM as a theoretical substitute for the CRCW PRAM.

Specific fully-parallel results

1. *The lowest common ancestor problem.* The preprocessing algorithm is fully-parallel, assuming that the Euler tour of the tree and the level of each vertex in the tree are given. A query can be

processed in constant time.

2. *The all nearest zero bit problem.* The algorithm is fully-parallel.

3. *The parentheses matching problem.* Assuming the level of nesting of each parenthesis is given, the algorithm is fully-parallel.

4. *Restricted domain merging.* The algorithm is fully-parallel. Results 3 and 4 can be derived from [BeV-90a] and [BeV-90] respectively similar to the fully-parallel algorithms here.

We elaborate on where our fully-parallel algorithms use the CRCW-bit new assumption. The algorithms work by mapping the input of size n into n bits. Then, given any constant d, we derive a value $x = O(I_d(n))$. The algorithms proceed by forming n/x groups of x bits each. Informally, our problem is then to pack all x bits of the same group into a single word and solve the original problem with respect to an input of size x in constant time. This packing is exactly where our almost fully-parallel CRCW PRAM algorithms fail to become fully-parallel and the CRCW-bit assumption is used. We believe that it is of theoretical interest to figure out ways for avoiding such packing and thereby get fully-parallel algorithms on a CRCW PRAM without the CRCW-bit assumption and suggest it as open problem.

A repeating motif in the present paper is putting restrictions on domain of problems. Perhaps our more interesting applications concern problems whose input domain is **not explicitly restricted**. However, as part of the design of our algorithms for these respective problems, we identified a few subproblems, whose input domains come restricted.

The rest of this paper is organized as follows. Section 2 describes the recursive *-tree data-structure and section 3 recalls a few basic problems and algorithms. In section 4 the algorithms for LCA and all nearest zero bit are presented. The almost fully parallel reducibility is presented in Section 5 and the last section discusses how to compute efficiently functions that are used in this manuscript.

2. The recursive *-tree data-structure

Let n be a positive integer. We define inductively a series of $\alpha(n)-1$ trees. For each $2 \leq m \leq \alpha(n)$ a balanced tree with n leaves, denoted $BT(m)$ (for Balanced Tree), is defined. For a given m, $BT(m)$ is a recursive tree in the sense that each of its nodes holds a tree of the form $BT(m-1)$.

The base of the inductive definition (see Figure 2.1). We start with the definition of the *-tree $BT(2)$. $BT(2)$ is simply a complete binary tree with n leaves.

The inductive step (see Figure 2.2). For m, $3 \leq m \leq \alpha(n)$, we define $BT(m)$ as follows. $BT(m)$ has n leaves. The number of levels in $BT(m)$ is $*I_{m-1}(n)+1$ $(= I_m(n)+1)$. The root is at level 1

and the leaves are at level $*I_{m-1}(n)+1$. Consider a node v at level $1 \leq l \leq *I_{m-1}(n)$ of the tree. Node v has $I_{m-1}^{(l-1)}(n)/I_{m-1}^{(l)}(n)$ children (we define $I_{m-1}^{(0)}(n)$ to be n). The total number of leaves in the subtree rooted at node v is $I_{m-1}^{(l-1)}(n)$. We refer to the part of the $BT(m)$ tree described so far as the *top recursion level* of $BT(m)$ (denoted for brevity $TRL-BT(m)$). In addition, node v contains recursively a $BT(m-1)$ tree. The number of leaves in this tree is exactly the number of children of node v in $BT(m)$.

In a nutshell, there is one key idea that enabled our algorithms to be as fast as they are. When the m'th tree $BT(m)$ is employed to guide the computation, **we invest $O(1)$ time on the top recursion level for $BT(m)$!** Since $BT(m)$ has m levels of recursion, this leads to a total of $O(m)$ time.

Similar computational structures appeared in a few contexts. See, [AS-87] and [Y-82] for generalized range-minima computations, [HS-86] for Davenport-Schinzel sequences and [CFL-83] for circuits.

3. Basics

We will need the following problems and algorithms.

The Euler tour technique

Consider a tree $T = (V,E)$, rooted at some vertex r. The Euler tour technique enables to compute several problems on trees in logarithmic time and optimal speed-up (see also [TV-85] and [Vi-85]). The technique is summerized below.

Step 1: For each edge $(v \rightarrow u)$ in T we add its anti-parallel edge $(u \rightarrow v)$. Let H denote the new graph.

Since the in-degree and out-degree of each vertex in H are the same, H has an Euler path that start and ends in the root r of T. Step 2 computes this path into a vector of pointers D, where for each edge e of H, $D(e)$ will have the successor edge of e in the Euler path.

Step 2: For each vertex v of H, we do the following:
(Let the outgoing edges of v be $(v \rightarrow u_0)$, $\cdots$, $(v \rightarrow u_{d-1})$.)
$D(u_i \rightarrow v) := (v \rightarrow u_{(i+1) mod\ d})$, for $i = 0, \cdots, d-1$. Now D has an Euler circuit. The "correction" $D(u_{d-1} \rightarrow r) := end-of-list$ (where the out-degree of r is d) gives an Euler path which starts and ends in r.

Step 3: In this step, we apply list ranking to the Euler path. This will result in ranking the edges so that the tour can be stored in an array. Similarly, we can find for each vertex in the tree its distance from the root. This distance is called the *level* of vertex v. Such applications of list ranking appear in [TV-85]. This list ranking can be performed in logarithmic time using an optimal number of processors, by [AM-88], [CV-86] or [CV-89].

Comments:

1. In Section 4 we assume that the Euler tour is given in an already ranked form. There we systematically replace each edge (u,v) in the Euler tour by the vertex v. We then add the root of the tree to the beginning of this new array. Suppose a vertex v has l children. Then v appears $l+1$ times in our array.

2. We note that while advancing from a vertex to its successor in the Euler tour, the level may either increase by one or decrease by one.

Finding the minimum for restricted-domain inputs

Input: Array $A=(a_1,a_2,\cdots,a_n)$ of numbers. The *restricted-domain* assumption: each a_i is an integer between 1 and n.

Finding the minimum. Find the minimum value in A.

Fich, Ragde and Wigderson [FRW-84] gave the following parallel algorithm for the restricted-domain minimum finding problem. It runs in $O(1)$ time using n processors. We use an auxiliary vector B of size n, that is all zero initially. Processor i ,$1 \le i \le n$, writes one into location $B(a_i)$. The problem now is to find the leftmost one in B. Partition B into $\sqrt{n}$ equal size subarrays. For each such subarray find in $O(1)$ time, using $\sqrt{n}$ processors if it contains a one. Apply the $O(1)$ time of Shiloach and Vishkin [SV-81] for finding the leftmost subarray of size $\sqrt{n}$ containing one, using n processors. Finally, reapply this latter algorithm for finding the index of the leftmost one in this subarray.

(*Remark 3.1.* This algorithm can be readily generalized to yield $O(1)$ time for inputs between 1 and p^c, where $c>1$ is a constant, as long as $p \ge n$ processors are used.)

Range-minima problem

Given an array $A=(a_1,a_2,\cdots,a_n)$ of n real numbers, preprocess the array so that for any interval $[a_i,a_{i+1},\cdots a_j]$, the minimum over the interval can be retrieved in constant time using a single processor.

We show how to preprocess A in constant time using $n^{1+\varepsilon}$ processors and $O(n^{1+\varepsilon})$ space, for any constant ε.

The preprocessing algorithm uses the following naive parallel algorithm for the range-minima problem. Allocate n^2 processors to each interval $[a_i\ ,\ a_{i+1}\ ,\ \cdots\ ,\ a_j]$ and find the minimum over the interval in constant time as in [SV-81]. This naive algorithm runs in constant time and uses n^4 processors and n^2 space.

The preprocessing algorithm. Suppose some constant ε is given.

The output of the preprocessing algorithm can be described by means of a balanced tree T with n leaves, where each internal node has $n^{\varepsilon/3}$ children. The root of the tree is at level one and the leaves are at level $3/\varepsilon+1$. Let v be an internal node and $u_1\ ,\ \cdots\ ,\ u_{n^{\varepsilon/3}}$ be its children. Each

internal node v will have the following data.

1. $M(v)$ - the minimum over its leaves (i.e. the leaves of its subtree).

2. For each $1 \le i < j \le n^{\varepsilon/3}$ the minimum over the range $\{M(u_i), M(u_{i+1}), \cdots, M(u_j)\}$.

This will require $O(n^{2/\varepsilon})$ space per internal node and a total of $O(n^{1+\varepsilon/3})$ space.

The preprocessing algorithm advances through the levels of the tree in $3/\varepsilon$ steps, starting from level $3/\varepsilon$. At each level, each node computes its data using the naive range-minima algorithm.

Each internal node uses $n^{4/\varepsilon}$ processors and $O(n^{2/\varepsilon})$ space and the whole algorithm uses $n^{1+\varepsilon}$ processors, $O(n^{1+\varepsilon/3})$ space and $O(3/\varepsilon)$ time.

Retrieval. Suppose we wish to find the minimum, $MIN(i,j)$ over an interval $[a_i, a_{i+1}, \cdots a_j]$.

Let $LCA(a_i,a_j)$ be the lowest common ancestor of leaf a_i and leaf a_j in the tree T. There are two possibilities.

(i) $LCA(a_i,a_j)$ is at level $3/\varepsilon$. The data at $LCA(a_i,a_j)$ gives $MIN(i,j)$.

(ii) $LCA(a_i,a_j)$ is at level $< 3/\varepsilon$. Let x be the number of edges in the path from a_i (or a_j) to $LCA(a_i,a_j)$ (i.e. $LCA(a_i,a_j)$ is at level $3/\varepsilon+1-x$). Using the tree T we can represent interval $[i,j]$ as a disjoint union of $2x-1$ intervals whose minimum were previously computed. Formally, let $r(i)$ denote the rightmost leaf of the parent of a_i in T and $l(j)$ denote the leftmost leaf of the parent of a_j in T. $MIN[i,j]$ is the minimum among three numbers.

1. $MIN[i,r(i)]$. The data at the parent of a_i gives this information.

2. $MIN[l(j),j]$. The data at the parent of a_j gives this information.

3. $MIN[r(i)+1,l(j)-1]$. Advance to level $3/\varepsilon$ of the tree to get this information recursively .

Complexity. Retrieval of $MIN(i,j)$ takes constant time: The first and second numbers can be looked up in $O(1)$ time. Retrieval of the third number takes $O(3/\varepsilon-1)$ time.

The range-minima problem was considered by [AS-87], [GBT-84] and [Y-82]. In the parallel context the problem was considered in [BBGSV-89].

4. LCA Algorithm

The input to this problem is a rooted tree $T=(V,E)$. Denote $n = 2|V|-1$. We assume that we are given a sequence of n vertices $A=[a_1, \ldots, a_n]$, which is the Euler tour of our input tree, and that we know for each vertex v its level, $LEVEL(v)$, in the tree.

Recall the range-minima problem defined in Section 3. Below we give a simple reduction from the LCA problem to a restricted-domain range-minima problem, which is an instance of the range-minima problem where *the difference between each two successive numbers for the range-minima problem is exactly one.* The reduction takes $O(1)$ time and uses n processors. An

algorithm for the restricted-domain range-minima problem is given later, implying an algorithm for the LCA problem.

4.1. Reducing the LCA problem into a restricted-domain range-minima problem

Let v be a vertex in T. Denote by $l(v)$ the index of the leftmost appearance of v in A and by $r(v)$ the index of its rightmost appearance. For each vertex v in T, it is easy to find $l(v)$ and $r(v)$ in $O(1)$ time and n processors using the following (trivial) observation:
$l(v)$ is where $a_{l(v)}=v$ and $LEVEL(a_{l(v)-1})=LEVEL(v)-1$.
$r(v)$ is where $a_{r(v)}=v$ and $LEVEL(a_{r(v)+1})=LEVEL(v)-1$.

The claims and corollaries below provide guidelines for the reduction.

Claim 1: Vertex u is an ancestor of vertex v iff $l(u) < l(v) < r(u)$.

Corollary 1: Given two vertices u and v, a single processor can find in constant time whether u is an ancestor of v.

Corollary 2: Vertices u and v are unrelated (namely, neither u is an ancestor of v nor v is an ancestor of u) iff either $r(u) < l(v)$ or $r(v) < l(u)$.

Claim 2. Let u and v be two unrelated vertices. (By Corollary 2, we may assume without loss of generality that $r(u) < l(v)$.) Then, the LCA of u and v is the vertex whose level is minimal over the interval $[r(u), l(v)]$ in A.

The reduction. Let $LEVEL(A) = [LEVEL(a_1), LEVEL(a_2),...,LEVEL(a_n)]$. Claim 2 shows that after performing the range-minima preprocessing algorithm with respect to $LEVEL(A)$, a query of the form $LCA(u,v)$ becomes a range minimum query. Observe that the difference between the level of each pair of successive vertices in the Euler tour (and thus each pair of successive entries in $LEVEL(A)$) is exactly one and therefore the reduction is into the restricted-domain range-minima problem as required.

Remark. [GBT-84] observed that the problem of preprocessing an array so that each range-minimum query can be answered in constant time (this is the range-minima problem defined in the previous section) is equivalent to the LCA problem. They gave a linear time algorithm for the former problem using an algorithm for the latter. This does not look very helpful: we know to solve the range-minima problem based on the LCA problem, and conversely, we know to solve the LCA problem based on the range-minima problem. Nevertheless, using the restricted domain properties of our range-minima problem we show that this cyclic relationship between the two problems can be broken and thereby, lead to a new algorithm.

4.2. The restricted-domain range-minima algorithm

We define below a restricted-domain range-minima problem which is slightly more general than the problem for *LEVEL*(A). The more general definition enables recursion in the algorithm below. The rest of this section shows how to solve this problem.

The restricted-domain range-minima problem

Input: Integer k and array $A=(a_1,a_2,\cdots,a_n)$ of integers, such that $|a_i-a_{i+1}| \le k$. In words, the difference between each a_i, $1 \le i < n$, and its successor a_{i+1} is at most k. The parameter k need not be a constant.

The range-minima problem. Preprocess the array $A=(a_1,a_2,\cdots,a_n)$ so that any query $MIN[i,j]$, $1 \le i < j \le n$, requesting the minimal element over the interval $[a_i, \cdots, a_j]$, can be processed quickly using a single processor.

Comment 1: We make the simplifying assumption that $\sqrt{k}$ is always an integer.

Comment 2. In case the minimum in the interval is not unique, find the minimal element in $[a_i, \cdots, a_j]$ whose index is smallest ("the leftmost minimal element"). Throughout this section, whenever we refer to a minimum over an interval, we will always mean the leftmost minimal element. Finding the leftmost minimal element (and not just the minimum value) will serve us later.

We start by constructing inductively a series of $\alpha(n)-1$ parallel preprocessing algorithms for our range-minima problem:

Lemma 4.2.1. The algorithm for $2 \le m \le \alpha(n)$ runs in cm time, for some constant c, using $nI_m(n) + \sqrt{k}n$ processors. The preprocessing algorithm results in a table. Using this table, any range-minimum query can be processed in cm time. In addition, the preprocessing algorithm finds explicitly all prefix-minima and suffix-minima, and therefore there is no need to do any processing for prefix-minima or suffix-minima queries.

Our optimal algorithms, whose efficiencies are given in Theorem 4.3.1, are derived from this series of algorithms.

We describe the series of preprocessing algorithms. We give first the base of the inductive construction and later the inductive step.

The base of the inductive construction (the algorithm for $m=2$)

In order to provide intuition for the description of the preprocessing algorithm for $m=2$ we present first its output and how the output can be used for processing a range-minimum query.

Output of the preprocessing algorithm for $m=2$:

(1) For each consecutive subarray $a_{j\log^3 n+1}, \cdots, a_{(j+1)\log^3 n}$, $0 \le j \le n/\log^3 n - 1$, we keep a table. The table enables constant time retrieval of any range-minimum query within the subarray.

(2) Array $B = b_1, \cdots, b_{n/\log^3 n}$ consisting of the minimum in each subarray.

(3) A complete binary tree $BT(2)$, whose leaves are $b_1, \cdots, b_{n/\log^3 n}$. Each internal node v of the tree holds an array P_v with an entry for each leaf of v. Consider prefixes that span between $l(v)$, the leftmost leaf of v, and a leaf of v. Array P_v has the minima over all these prefixes. Node v also holds a similar array S_v. For each suffix, that spans between a leaf v and $r(v)$, the rightmost leaf of v, array S_v has its minimum.

(4) Two arrays of size n each, one contains all prefix-minima and the other all suffix-minima with respect to A.

Lemma 4.2.2. Let m be 2. Then $I_m(n) = \log n$. The preprocessing algorithm will run in $2c$ time for some constant c using $n\log n + \sqrt{k}n$ processors. The retrieval time of a query $MIN[i,j]$ is $2c$.

How to retrieve a query $MIN[i,j]$ in constant time?

There are two possibilities.

(i) a_i and a_j belong to the same subarray (of size $\log^3 n$). $MIN(i,j)$ is computed in $O(1)$ time using the table that belongs to the subarray.

(ii) a_i and a_j belong to different subarrays. We elaborate below on possibility (ii).

Let $right(i)$ denote the rightmost element in the subarray of a_i and $left(j)$ denote the leftmost element in the subarray of a_j. $MIN[i,j]$ is the minimum among three numbers.

1. $MIN[i,right(i)]$, the minimum over the suffix of a_i in its subarray.
2. $MIN[left(j),j]$, the minimum over the prefix of a_j in its subarray.
3. $MIN[right(i)+1,left(j)-1]$.

The retrieval of the first and second numbers is similar to possibility (i) above. Denote $i_1 = \left\lceil i/\log^3 n \right\rceil + 1$ and $j_1 = \left\lceil j/\log^3 n \right\rceil - 1$. We discuss retrieval of the third number. This is equal to finding the minimum over interval $[b_{i_1}, \cdots, b_{j_1}]$ in B, which is denoted $MIN_B[i_1,j_1]$.

Let x be the lowest common ancestor of b_{i_1} and b_{j_1}, x_1 be the child of x that is an ancestor of b_{i_1} and x_2 be the child of x that is an ancestor of b_{j_1}.

$MIN_B[i_1,j_1]$ is the minimum among two numbers:

1. $MIN_B[i_1,r(x_1)]$, the minimum over the suffix of b_{i_1} in x_1. We get this from S_{x_1}.
2. $MIN_B[l(x_2),j_1]$, the minimum over the prefix of b_{j_1} in x_2. We get this from P_{x_2}.

It remains to show how to find x, x_1 and x_2 in constant time. It was observed in [HT-84] that the lowest common ancestor of two leaves in a complete binary tree can be found in constant time using one processor. (The idea is to number the leaves from 0 to $n-1$. Given two leaves i_1 and j_1, it suffices to find the most significant bit in which the binary representation of i_1 and j_1 are different, in order to get their lowest common ancestor.) Thus x (and thereby also x_1 and x_2) can be found in constant time. Constant time retrieval of $MIN(i,j)$ query follows.

The preprocessing algorithm for $m=2$

Step 1. Partition A into subarrays of $\log^3 n$ elements each. Allocate $\log^4 n$ processors to each subarray and apply the preprocessing algorithm for range-minima given in Section 3 (for $\varepsilon = 1/3$). This uses $\log^4 n$ processors and $O(\log^3 n \log^{1/3} n)$ space per subarray and $n \log n$ processors and $o(n \log n)$ space overall.

Step 2. Take the minimum in each subarray to build array B of size $n/\log^3 n$. The difference between two successive elements in B is at most $k \log_k n$.

Step 3: Build $BT(2)$, a complete binary tree, whose leaves are the elements of B. For each internal node v of $BT(2)$ we keep an array. The array consists of the values of all leaves in the subtree rooted at v. So, the space needed is $n/\log^3 n$ per level and $n/\log^2 n$ for all levels of the tree. We allocate to each leaf at each level $\sqrt{k} \log^2 n$ processors and the total number of processors used is thus $\sqrt{k} n$.

Step 4: For each internal node, find the minimal element over its array. If the minimal element is not unique, the leftmost one is found. We apply the constant time algorithm mentioned in Remark 3.1. Consider an internal node of size r. After subtracting the first element of the array from each of its elements, we get an array whose elements range between $-kr \log^3 n$ and $kr \log^3 n$. The size of the range, which is $2kr \log^3 n + 1$, does not exceed the square of number of processors, which is $r\sqrt{k} \log^2 n$, and the algorithm of Remark 3.1 can be applied.

Step 5: For each internal node v we compute P_v (S_v is computed similarly): That is, for each leaf b_i of v, we need to find $MIN_B[l(v),i]$ (that is, the minimum over the prefix of b_i in v). For this, the minimum among the following list of (at most) $\log n + 1$ numbers is computed. Denote the level of v in the binary tree by $level(v)$. Each level l, $level(v) < l \leq \log n + 1$, of the tree contributes (at most) one number. Let u denote the ancestor at level $l-1$ of b_i. Let u_1 and u_2 denote the left and right children of u respectively. If b_i belongs to (the subtree rooted at) u_2 then level l contributes the minimum over u_1. If b_i belongs to u_1 then level l does not contribute anything (actually, level l contributes a large default value so that the minimum computation is not affected). Finally, b_i is also included in the list. This minimum computation can be done in constant time using $\log^2 n$ processors by the algorithm of [SV-81]. Note that all prefix-minima and all

suffix-minima of B are computed (in the root) in this step.

Step 6. For each a_i we find its prefix minimum and its suffix minimum with respect to A using one processor in constant time. Let b_j be the minimum representing the subarray of size $\log^3 n$ containing a_i. The minimum over the prefix of a_i with respect to A is the minimum between the prefix of b_{j-1} with respect to B and the minimum over the prefix of a_i with respect to its subarray.

This completes the description of the inductive base: Items (1), (2), (3) and (4) of the output were computed (respectively) in steps 1, 2, 5 and 6 above.

Complexity of the inductive base. $O(1)$ time using $n \log n + \sqrt{k}\, n$ processors.

Lemma 4.2.2 follows.

The inductive step

The algorithm is presented in a similar way to the inductive base.

Output of the m'th preprocessing algorithm

(1) For each consecutive subarray $a_{jI_m^3(n)+1}, \cdots, a_{(j+1)I_m^3(n)}$, $0 \le j \le n/I_m^3(n)-1$, we keep a table. The table enables constant time retrieval of any range-minimum query within the subarray. (*Comment.* The notation $I_m^3(n)$ means $(I_m(n))^3$ where $I_m(n)$ is defined earlier.)

(2) Array $B = b_1, \cdots, b_{n/I_m^3(n)}$ consisting of the minimum in each subarray.

(3) $TRL\text{-}BT(m)$, the top recursion level of (the recursive *-tree) $BT(m)$, whose leaves are $b_1, \cdots, b_{n/I_m^3(n)}$. Each internal node v of $TRL\text{-}BT(m)$ holds an array P_v and array S_v with an entry for each leaf of v. These arrays hold (as in the binary tree for m=2) prefix-minima and suffix-minima with respect to the leaves of v.

(4) Let $u_1, \cdots, u_y$ be the children of v, an internal node of $TRL\text{-}BT(m)$. Denote by $MIN(u_i)$ be the minimum over the leaves of node u_i, $1 \le i \le y$. Each such node v has recursively the output of the $m-1$'th preprocessing algorithm with respect to the input $MIN(u_1), \cdots, MIN(u_y)$.

(5) Two arrays of size n each, one contains all prefix-minima and the other all suffix-minima with respect to A.

How to retrieve a query $MIN[i,j]$ in cm time?

We distinguish two possibilities.

(i) a_i and a_j belong to the same subarray (of size $I_m^3(n)$). $MIN(i,j)$ is computed in $O(1)$ time using the table that belongs to the subarray.

(ii) a_i and a_j belong to different subarrays. We elaborate on possibility (ii).
Again, let $right(i)$ denote the rightmost element in the subarray of a_i and $left(j)$ denote the leftmost element in the subarray of a_j. $MIN[i,j]$ is the minimum among three numbers.

1. $MIN[i,right(i)]$.
2. $MIN[left(j),j]$.
3. $MIN[right(i)+1,left(j)-1]$.

The retrieval of the first and second numbers is similar to possibility (i) above. Denote $i_1 = \left\lceil i/I_m^3(n) \right\rceil + 1$ and $j_1 = \left\lceil j/I_m^3(n) \right\rceil - 1$. Retrieval of the third number is equal to finding the minimum over interval $[b_{i_1}, \cdots, b_{j_1}]$ in B which is denoted $MIN_B[i_1,j_1]$.

Let x be the lowest common ancestor of b_{i_1} and b_{j_1} in $TRL-BT(m)$, $x_{\beta(i_1)}$ be the child of x that is an ancestor of b_{i_1} and $x_{\beta(j_1)}$ be the child of x that is an ancestor of b_{j_1}. $MIN_B[i_1,j_1]$ is (recursively) the minimum among three numbers.

1. $MIN_B[i_1,r(x_{\beta(i_1)})]$, the minimum over the suffix of b_{i_1} in $x_{\beta(i_1)}$. We get this from $S_{x_{\beta(i_1)}}$.
2. $MIN_B[l(x_{\beta(j_1)}),j_1]$, the minimum over the prefix of b_{j_1} in $x_{\beta(j_1)}$. We get this from $P_{x_{\beta(j_1)}}$.
3. $MIN_B[r(x_{\beta(i_1)})+1,l(x_{\beta(j_1)})-1]$. This will be recursively derived from the data at node x.

The first two numbers are precomputed in $TRL-BT(m)$. The recursive definition of the third number implies that $MIN_B[i_1,j_1]$ is actually the minimum among $4(m-1)-2$ precomputed numbers. Therefore, in order to show that retrieval of $MIN[i,j]$ takes time proportional to m, as claimed, it remains to explain how to find the nodes x, $x_{\beta(i_1)}$ and $x_{\beta(j_1)}$ in constant time using one processor. This is done below.

We first note that for each leaf of $TRL-BT(m)$, finding the child of the root which is its ancestor needs constant time using one processor. Given two leaves of $TRL-BT(m)$ consider their ancestors among the children of the root. If these ancestors are different, we are done. Suppose these ancestors are the same.

Each child of the root has $I_{m-1}(n/I_m^3(n)) \leq I_{m-1}(n)$ leaves. Observe that for $TRL-BT(m)$ the same subtree structure is replicated at each child of the root. For each pair of two leaves u and v of the generic subtree structure, we will compute three items into a table: (1) their lowest common ancestor w; (2) the child f of w which is an ancestor of u; (3) the child g of w which is an ancestor of v. The size of the table is only $O(I_{m-1}^2(n))$.

It remains to show how the table is computed. Consider an internal node w of the tree and suppose that its rooted subtree has r leaves. At node w each pair of leaves u, v is allocated to a processor. The processor determines in constant time if w is the LCA of u and v. This is done by finding whether the child of w which is an ancestor of u, denoted f, and the child of w which

is an ancestor of v are different. If yes, then w, f and g are as required for the table. The number of processors needed for computing the table is $O(I^2_{m-1}(n))$.

The preprocessing algorithm for m

Inductively, we assume that we have an algorithm that preprocesses the array $A=(a_1,a_2,\cdots,a_n)$ for the range-minima problem in $c(m-1)$ time using $nI_{m-1}(n)+\sqrt{k}n$ processors, where c is a constant; and that following this preprocessing any $MIN[i,j]$ query can be answered in $c(m-1)$ time. We construct an algorithm that solves the range-minima problem in $c_1+c(m-1)$ time for some constant c_1, using $nI_m(n)+\sqrt{k}n$ processors. We have already shown that a query can be answered in $c_2 m$ time for some constant c_2. Selecting initially $c > c_1$ and $c > c_2$ implies that the algorithm runs in cm time using $nI_m(n)+\sqrt{k}n$ processors and that a query can be answered in cm time.

Step 1. Partition A into subarrays of $I^3_m(n)$ elements each. Allocate $I^4_m(n)$ processors to each subarray and apply the preprocessing algorithm for range-minima given in Section 3 (for $\varepsilon = 1/3$). This uses $I^4_m(n)$ processors and $O(I^3_m(n)I^{1/3}_m(n))$ space per subarray and $nI_m(n)$ processors and $o(nI_m(n))$ space overall.

Step 2. Take the minimum in each subarray to build array B of size $n/I^3_m(n)$. The difference between two successive elements in B is at most $kI^3_m(n)$.

Step 3: Build $TRL-BT(m)$, the upper level of a $BT(m)$ tree whose leaves are the elements of B. Each internal node of $TRL-BT(m)$, whose rooted tree has r leaves, has $r/I_{m-1}(r)$ children. For each such internal node v of $TRL-BT(m)$ we keep an array. The array consists of the values of the r leaves of the subtree rooted at v. $TRL-BT(m)$ will have $*I_{m-1}(n/I^3_m(n))+1 \le I_m(n)+1$ levels. So, the space needed is $n/I^3_m(n)$ per level and $O(n/I^2_m(n))$ for all levels of $TRL-BT(m)$. We allocate to each leaf at each level $1+\sqrt{k}I^2_m(n)$ processors and the total number of processors used is thus $n/I^2_m(n)+\sqrt{k}n$ (which is less than $nI_m(n)+\sqrt{k}n$).

Step 4.1: For each internal node of $TRL-BT(m)$, find the minimum over its array. The difference between the minimum value and the maximum value in an array never exceeds the square of its number of processors and we apply the constant time algorithm mentioned in Remark 3.1 as in Step 4 of the inductive base algorithm.

Step 4.2: We focus on internal node v having r leaves in $TRL-BT(m)$. Each of its $r/I_{m-1}(r)$ children contributes its minimum and we preprocess these minima using the assumed algorithm for $m-1$. The difference between adjacent elements is at most $kI_{m-1}(r)I^3_m(n)$. Thus, this computation takes $c(m-1)$ time using $r+\sqrt{kI^3_m(n)}r$ processors. (To see this, simplify $\frac{r}{I_{m-1}(r)}I_{m-1}(r)+\sqrt{kI_{m-1}(r)I^3_m(n)}\,\frac{r}{I_{m-1}(r)}$, the processor count term for this problem, into $r+\sqrt{kI^3_m(n)}\,r/\sqrt{I_{m-1}(r)}$ which is less than $r+\sqrt{kI^3_m(n)}r$ processors.) This amounts to

$n/I_m^3(n)+\sqrt{I_m^3(n)}\,n/I_m^3(n)$ per level or a total of $n/I_m^2(n)+\sqrt{k}\,n/\sqrt{I_m(n)}$ processors which is less than $n/I_m^2(n)+\sqrt{k}\,n$.

Step 5: For each internal node v we compute P_v (S_v is computed similarly). That is, for each leaf b_i of v we need to find $MIN_B[l(v),i]$, the minimum over the prefix of b_i with respect to the leaves of node v. For this, the minimum among the following list of at most $*I_{m-1}(n)+1 = I_m(n)+1$ numbers is computed: Each level l, $level(v) < l \le I_m(n)+1$, of the tree contributes (at most) one number. Let u denote the ancestor at level $l-1$ of b_i and let $u_1, \cdots, u_y$ denote its children, which are at level l. Suppose u_j, $j>1$ is an ancestor of b_i. We take the prefix-minimum over the leaves of $u_1, \cdots, u_{j-1}$. This prefix-minimum is computed in the previous step (by the assumed algorithm for $m-1$). If u_1 is the ancestor of b_i then level l contributes a large default value (as in Step 5 of the inductive base algorithm). Finally, b_i is also added to the list. This minimum computation can be done in constant time using $I_m^2(n)$ processors (by the algorithm of [SV-81]). Note that all prefix-minima and all suffix-minima with respect to B are computed (in the root) in this step.

Step 6. For each a_i we find its prefix minimum and its suffix minimum with respect to A using one processor in constant time. This is similar to Step 6 of the inductive base.

This completes the description of the inductive step: Items (1), (2), (3), (4) and (5) of the output were computed (respectively) in steps 1, 2, 5, 4.2 and 6 above.

Complexity of the inductive step

In addition to application of the inductively assumed algorithm, steps 1 through 6 take constant time using $nI_m(n) + \sqrt{k}\,n$ processors. This totals cm time using $nI_m(n) + \sqrt{k}\,n$ processors.

Together with Lemma 4.2.2, Lemma 4.2.1 follows.

From recursion to algorithm

The recursive procedure in Lemma 4.2.1 translates easily into a constructive parallel algorithm where the instructions for each processor at each time unit are available. For such translation issues such as processor allocation and computation of certain functions need to be taken into account. Since $TRL-BT(m)$ is balanced, allocating processors in the algorithm above can be done in constant time if the following functions are precomputed: (a) $I_m(x)$ for $1 \le x \le n$ and (b) $I_{m-1}^{(i)}(x)$ for $1 \le x \le n$ and $1 \le i \le I_m(x)$. These same functions suffice for all other computations above. The functions will be computed and stored in a table at the beginning of the algorithm. The last section discusses their computation.

4.3. The optimal parallel algorithms

In this subsection, we show how to derive a series of optimal parallel algorithms from the series of algorithms described in Lemma 4.2.1. Theorem 4.3.1 gives a general trade-off result between running time of the preprocessing algorithm and retrieval time. Corollary 4.3.1 emphasizes results where the retrieval time for a query is constant. Corollary 4.3.2 points at an interesting trade-off instance where the retrieval time bound is increased to $O(\alpha(n))$ and the preprocessing algorithm runs in $O(\alpha(n))$ (i.e., it become almost fully-parallel).

Theorem 4.3.1. Consider the range-minima problem, where k, the bound on the difference between two successive elements in A, is constant. For each $2 \leq m \leq \alpha(n)$, we present a parallel preprocessing algorithm whose running time is $O(I_m(n))$ using an optimal number of processors. The retrieval time of a query is $O(m)$.

Corollary 4.3.1. When m is constant the preprocessing algorithm runs in $O(I_m(n))$ time using $n/I_m(n)$ processors. Retrieval time is $O(1)$.

Corollary 4.3.2. When $m = \alpha(n)$ the preprocessing algorithm runs in $O(\alpha(n))$ time using $n/\alpha(n)$ processors. Retrieval time is $O(\alpha(n))$.

We describe below the optimal preprocessing algorithm for m as per Theorem 4.3.1.

Step 1. Partition A into subarrays of $I_m^2(n)$ elements each, allocate $I_m(n)$ processors to each subarray and find the minimum in the subarray. This can be done in $O(I_m(n))$ time.

Put the $n/I_m^2(n)$ minima into an array B.

Step 2. Out of the series of preprocessing algorithms of Lemma 4.2.1 apply the algorithm for m to B, where k', the difference between two successive elements of B is $O(I_m^2(n))$. This will take $O(m)$ time using $\sqrt{k'}\, n/I_m^2(n)+n/I_m(n)$ processors. This can be simulated in $O(m)$ time using $n/I_m(n)$ processors.

Step 3. Preprocess each subarray of $I_m^2(n)$ elements so that a range-minimum query within the subarray can be retrieved in $O(1)$ time. This is done using the following parallel variant of the range-minima algorithm of [GBT-84].

Range-minimum: a parallel variant of GBT's algorithm

Consider the general range-minima problem as defined in Section 3, with respect to an input array $C=(c_1,c_2,\cdots,c_n)$. We overview a preprocessing algorithm that runs in $O(\sqrt{n})$ time using $\sqrt{n}$ processors, so that a range-minimum query can be processed in constant time.

(1) Partition array C into $\sqrt{n}$ subarrays $C_1, \cdots, C_{\sqrt{n}}$ each with $\sqrt{n}$ elements.

(2) Apply the linear time serial algorithm of [GBT-84] separately to each C_i, taking $O(\sqrt{n})$ time using $\sqrt{n}$ processors.

(3) Let $\overline{c_i}$ be the minimum over C_i. Apply GBT's algorithm to $\overline{C} = (\overline{c_1}, \cdots, \overline{c_{\sqrt{n}}})$ in $O(\sqrt{n})$ time using a single processor.

It should be clear that any range-minimum query with respect to C can be retrieved in constant time by at most three queries with respect to the tables built by the above applications of GBT's algorithm.

Complexity of the preprocessing algorithm. $O(I_m(n))$ time using $n/I_m(n)$ processors. Retrieval of a range-minimum query will take $O(m+1)$ time which is $O(m)$ time.

Theorem 4.3.1 follows.

4.4. The fully-parallel algorithms

Consider the restricted-domain range-minima problem where k, the bound on the difference between adjacent elements, is constant. In this subsection, we present a fully-parallel preprocessing algorithm for the problem on a CRCW-bit PRAM that provides for constant time processing of a query. Theorem 4.4.1 gives the general result being achieved in this subsection including tradeoffs among parameters. Corollary 4.4.1 summarizes the fully-parallel result.

Let d be an integer $2 < d \leq \alpha(n)$. The model of parallel computation is the CRCW-bit PRAM with the assumption that up to $I_d(n)$ processors may write simultaneously into different bits of the same memory word.

Theorem 4.4.1 The preprocessing algorithm takes $O(d)$ time using n processors. The retrieval time for a query $MIN(i,j)$ is $O(d)$.

Remark. Theorem 4.4.1 represent a tradeoff between the time for the preprocessing algorithm and query retrieval on one hand and the number of processors that may write simultaneously into different bits of the same memory word on the other hand.

Corollary 4.4.1. For a constant d, the algorithm is fully-parallel and query retrieval time is constant.

Step 1. Partition A into $n/I_d(n)$ subarrays of $I_d(n)$ elements each. For each subarray, find the minimum in $O(1)$ time and $I_d(n)$ processors. For this we apply the constant time algorithm mentioned in Remark 3.1 as in Step 4 of the inductive base algorithm.

Put the $n/I_d(n)$ minima into an array B. The difference between two successive elements in B is at most $kI_d(n)$.

Step 2. Out of the series of preprocessing algorithms of Lemma 4.2.1 apply the algorithm for d to B, where k', the difference between two successive elements of B is $O(I_d(n))$. This will take $O(d)$ time and $\sqrt{k'}n/I_d(n)+n$ processors and can be simulated in $O(d)$ time using n processors.

Suppose we know to retrieve a range-minimum query within each of the subarrays of size $I_d(n)$ in constant time. It should be clear how a query $MIN(i,j)$ can then be retrieved in $O(d)$ time. Theorem 4.4.1 would follow.

Thus, it remains to show how to preprocess the subarrays of size $I_d(n)$ in constant time such that a range-minimum query within a subarray can be retrieved in constant time. The preprocessing of these subarrays is done in steps 3.1, 3.2 and 3.3.

Step 3.1. For each subarray, subtract the value of its first element from each element of the subarray.

Observe that following this subtraction the value of the first element is zero and the difference between each pair of successive elements remains at most k. Step 3.2 constructs a table with the following information: For any $I_d(n)$-tuple $(c_1,...,c_{I_d(n)})$, where $c_1=0$ and the difference between each pair of successive c_i values is at most k, the table has an entry. This entry gives all $I_d(n)(I_d(n)-1)/2$ range-minima with respect to this $I_d(n)$-tuple.

Step 3.2. All n processors together build a table. Each entry of the table corresponds to one possible allocation of values to the $I_d(n)$-tuple. The entry will provide all $I_d(n)(I_d(n)-1)/2$ range-minima for this allocation.

Observe that the number of possible allocations is $(2k+1)^{I_d(n)-1}$. To see this, we note that each possible allocation can be characterized by a sequence of $I_d(n)-1$ numbers taken from $[-k,k]$. This will indeed be the number of entries in our table. Using n processors (or even less) the table can be built in $O(1)$ time.

Step 3.3. The only difficulty is to identify the table entry for our $I_d(n)$-tuple $c_1, \cdots, c_{I_d(n)}$, since once we reach the entry, the table will already provide the desired range-minima. We allocate to each subarray $I_d(n)$ processors. For each subarray, we have a word in our shared memory with $(I_d(n)-1)\log(2k+1)$ bits. Processor i , $1 < i \leq I_d(n)$, will write $c_i - c_{i-1}$ (which is a number from $[-k,k]$) starting in bit number $(i-2)\log(2k+1)$ of the word belonging to its subarray (bit zero being the least significant). As a result this word will have a sequence of numbers from $[-k,k]$ that yields the desired entry in our table. Note that exactly $I_d(n)$ processors write to different bits of the same memory word.

Theorem 4.4.1 follows.

4.5. The all nearest zero bit problem

The following corollary of theorems 4.3.1 and 4.4.1 is needed for Section 5.

Corollary 4.5.1. The all nearest zero bit problem is almost fully-parallel. On the CRCW-bit PRAM the all nearest zero bit problem is fully-parallel.

Proof. Recall that the algorithm for the restricted-domain range-minima problem computes all suffix-minima. Recall also that in case that the minimum over an interval is not unique, the left-most minimum is found. Thus if we apply the restricted-domain range-minima algorithm (with difference between successive elements at most one) with respect to A then the minimum over the suffix of entry $i+1$ gives the nearest zero to the right of entry i. Thus, the all nearest zero bit is actually an instance of the restricted-domain range-minima problem (with difference between successive elements at most one). It follows that the almost fully-parallel and the fully-parallel algorithms for the latter apply for the all nearest zero bit problem as well.

5. Almost fully-parallel reducibility

We demonstrate how to use the * -tree data structure for reducing a problem A into another problem B by an almost fully parallel algorithm. We apply this reduction for deriving a parallel lower bound for problem A from a known parallel lower bound for problem B.

Given a convex polygon with n vertices, the *all nearest neighbors (ANN)* problem is to find for each vertex of the polygon its nearest (Euclidean) neighbor.

Theorem 5.1. Any CRCW PRAM algorithm for the ANN problem that uses $O(n \log^c n)$ (for any constant c) processors needs $\Omega(\log\log n)$ time.

Proof. We give below an almost fully-parallel reduction from the problem of merging two sorted lists of length n each to the ANN problem with $O(n)$ vertices. This reduction together with the following lemma imply the Theorem.

Lemma. Merging two sorted lists of length n each using $O(n \log^c n)$ (for any constant c) processors on a CRCW PRAM needs $\Omega(\log\log n)$ time.
A remark in [ScV-88a] implies that Borodin and Hopcroft's ([BHo-85]) lower bound for merging in a parallel comparisons model can be extended to yield the Lemma.

Proof of Theorem 5.1 (continued).

The reduction (see Figure 5.1):
Let $A=(a_1,a_2,\cdots,a_n)$ and $B=(b_1,b_2,\cdots,b_n)$ be two increasing lists of numbers that we wish to merge. Assume, without loss of generality, that the numbers are integers and that $a_1 = b_1$, $a_n = b_n$. (The lower bound for merging assumes that the numbers are integers.) Consider the following *auxiliary* problem: For each $1 \le i \le n$, find the minimum index j such that $b_j > a_i$. The position of a_i in the merged list is $i+j-1$ and therefore an algorithm for the auxiliary problem (together with a similar algorithm for finding the positions of the b_i numbers in the merged list) suffices for the merging problem.

We give an almost fully-parallel reduction from the auxiliary problem to the ANN problem with respect to the following convex polygon.

Let $(c_1, c_2, \cdots, c_{2n-1}) = (a_1, \frac{a_1+a_2}{2}, a_2, \frac{a_2+a_3}{2}, \cdots, a_{n-1}, \frac{a_{n-1}+a_n}{2}, a_n)$. The numbers $c_1, c_2, \cdots, c_{2n-1}$ form an increasing list. The convex polygon is $(c_1,0), (c_2,0), \cdots, (c_{2n-1},0), (b_n,1/4), (b_{n-1},1/4), \cdots, (b_1,1/4)$.

In [ScV-88a] a similar construction is given and the lower bound proof then follows by (non trivial) Ramsey theoretic arguments.

Let $D[1, \cdots, 2n-1]$ be a binary vector. Each vertex $(c_l,0)$ finds its nearest neighbor with respect to the convex polygon (using a 'supposedly existing' algorithm for the ANN problem) and assigns the following into vector D.

If the nearest vertex is of the form $(b_k,1/4)$
then $D(l) := 0$
else $D(l) := 1$

Next we apply to vector D the almost fully-parallel algorithm for the nearest zero bit problem of Subsection 4.5. Finally, we show how to solve our auxiliary problem with respect to every element a_i. We break into two cases concerning the nearest neighbor of $(a_i,0)$ ($=(c_{2i-1},0)$).

Case (i). The nearest neighbor of $(a_i,0)$ is a vertex $(b_\alpha,1/4)$. Then, the minimum index j such that $b_j > a_i$ is either α or $\alpha+1$. A single processor can determine the correct value of j in $O(1)$ time.

Case (ii). Otherwise. Then, $D(2i-1)=1$. The nearest zero computation gives the smallest index $k > 2i-1$ for which $D(k)=0$. Let the nearest neighbor of $(c_k,0)$ be $(b_\alpha,1/4)$. Then $j=\alpha$ is the minimum index for which $b_j > a_i$.

6. Computing various functions

We need to compute certain functions during our algorithms. For each $2 < m \leq \alpha(n)$ we need $I^{(i)}_{m-1}(n)$ for all $1 \leq i \leq I_m(n)$. Fortunately, the function parameters that we are actually concerned with are small, relative to n. For instance, in order to compute $I_3(n) = \log^* n$, it is enough to compute $\log^*(\text{loglog}\, n)$ since $\log^* n = \log^*(\text{loglog}\, n)+2$.

We show only how to compute $I_m(n)$ for each $2 \leq m \leq \alpha(n)$. Computation of $I^{(i)}_{m-1}(n)$ for $2 < m \leq \alpha(n)$ and all $1 \leq i \leq I_m(n)$ is similar. Our computation works by induction on m.
The inductive hypothesis: Let $x = \text{loglog}\, n$. We know to compute the following values in $O(m-1)$ time using $o(n/\alpha(n))$ processors: (1) $I_{m-1}(n)$; and (2) $I_{m-1}(y)$ for all $1 \leq y \leq \text{loglog}\, n$.

We show the inductive claim (the claim itself should be clear), assuming the inductive hypothesis. The inductive base is given later. First, we describe (informally) the computation of $I_m(x)$ in $O(1)$ (additional) time using $o(n/\alpha(n))$ processors. Consider all permutations of the numbers $1, \cdots, x$. The number of these permutations is (much) less than n. The idea is to

identify a permutation that provides the sequence $[x\ , I_{m-1}(x)\ , I^{(2)}_{m-1}(x)\ , \cdots , I^{(k)}_{m-1}(x) = 1\ , \cdots\ ]$. So if $I_{m-1}(p_i) = p_{i+1}$ for all $0 \le i \le k-1$ we conclude $I_m(x) = k$. We can check this condition in $O(1)$ time using x processors per permutation, using the ability of the CRCW PRAM to find the AND of x bits in $O(1)$ time. The total number of processors is $o(n/\alpha(n))$. We make two remarks: (1) Computing $I_m(y)$ for all $1 \le y \le \text{loglog}\, n$, the rest of the inductive claim, in $O(1)$ time using $o(n/\alpha(n))$ processors is similar. (2) there are easy ways for finding all permutations in O(1) time using the number of available processors.

We finish by showing the inductive base. We compute log$n$ in $O(1)$ time and $o(n/\alpha(n)$ processors as follows. If n is given in a binary prepresentation then the index of the leftmost one is log$n$. Following [FRW-84]), this can be computed in $O(1)$ using as many processors as the number of bits of a number. By iterating this we get $\log^{(2)}n$. Finally, we find log$y$ for all $1 \le y \le \text{loglog}\, n$. The number of processors used for this computation is $o(n/\alpha(n))$.

Acknowledgments

We are grateful to Pilar de la Torre and to Baruch Schieber for fruitful discussions and helpful comments.

References

[AILSV-88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber and U. Vishkin, "Parallel construction of a suffix tree with applications", *Algorithmica* 3 (1988), 347-365.

[AM-88] R.J. Anderson and G.L. Miller, "Deterministic parallel list ranking", *Proc. 3rd AWOC* (1988), 81-90.

[AMW-88] R.J. Anderson, E.W. Mayr and M. K. Warmuth, "Parallel approximation algorithms for bin packing", *Information and Computation,* 82 (1989), 262-277.

[AS-87] N. Alon and B. Schieber, "Optimal preprocessing for answering on-line product queries", TR 71/87, The Moise and Frida Eskenasy Institute of Computer Science, Tel Aviv University (1987).

[BBGSV-89] O. Berkman, D. Breslauer, Z. Galil, B. Schieber and U. Vishkin, "Highly Parallelizable Problems", *Proc. 21th ACM Symp. on Theory of Computing* (1989), 309-319.

[BHa-87] P. Beame and J. Hastad, "Optimal Bound for Decision Problems on the CRCW PRAM", *Proc. 19th ACM Symp. on Theory of Computing* (1987), 83-93.

[BHo-85] A. Borodin and J.E. Hopcroft, "Routing, merging, and sorting on parallel models of comparison", *J. of Comp. and System Sci.*, 30 (1985), 130-145.

[BSV-88] O. Berkman, B. Schieber and U. Vishkin, "Some doubly logarithmic parallel algorithms based on finding all nearest smaller values", UMIACS-TR-88-79, University of Maryland institute for advanced computer studies (1988).

[BV-85] I. Bar-On and U. Vishkin, "Optimal parallel generation of a computation tree form", *ACM Trans. on Prog. Lang. and Systems*, 7 (1985), 348-357.

[BeV-90] O. Berkman and U. Vishkin, "On parallel integer merging", UMIACS-TR-90-15, University of Maryland Inst. for Advanced Comp. Studies (1990).

[BeV-90a] O. Berkman and U. Vishkin, "Almost fully-parallel parentheses matching", in preparation.

[CFL-83] A.K. Chandra, S. Fortune and R. Lipton, "Unbounded fan-in circuits and associative functions", *Proc. 15th ACM Symp. on Theory of Computing* (1983), 52-60.

[CV-86] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems", *Proc. 27th Annual Symp. on Foundations of Computer Science* (1986), 478-491.

[CV-86a] R. Cole and U. Vishkin, "Deterministic coin tossing with applications to optimal parallel list ranking", *Information and Control*, 70 (1986), 32-53.

[CV-89] R. Cole and U. Vishkin, "Faster optimal prefix sums and list ranking", *Information and Computation* 81,3 (1989), 334-352.

[DS-83] E. Dekel, and S. Sahni, "Parallel generation of postfix and tree forms", *ACM Trans. on Prog. Languages and Systems*, 5 (1983), 300–317.

[FRT-89] D. Fussell, V. Ramachandran and R. Thurimella, "Finding triconnected components by local replacements", *Proc. 16th ICALP* (1989).

[FRW-84] F.E. Fich, R.L. Ragde and A. Wigderson, "Relations between concurrent-write models of parallel computation" (preliminary version), *Proc. 3rd ACM Symp. on Principles of Distributed Computing* (1984), 179-189. Also, *SIAM J. Comput.* 17,3 (1988), 606-627.

[GBT-84] H.N. Gabow, J.L. Bentley and R.E. Tarjan, "Scaling and related techniques for geometry problems", *Proc. 16th ACM Symp. on Theory of Computing* (1984), 135-143.

[H-86] J. Hastad, "Almost optimal lower bounds for small depth circuits", *Proc. 18th ACM Symp. on Theory of Computing* (1986), 6-20.

[HS-86] S. Hart and M. Sharir, "Non linearity of Davenport-Schinzel sequences and generalized path compression schemes", *Combinatorica*, 6,2 (1986), 151-177.

[HT-84] D. Harel and R.E. Tarjan, "Fast algorithms for finding nearest common ancestors", *SIAM J. Comput.*, 13 (1984), 338-355.

[Kr-83] C.P. Kruskal, "Searching, merging, and sorting in parallel computation", *IEEE Trans. on Computers*, C-32 (1983), 942-946.

[LF-80] R.E. Ladner and M.J. Fischer, "Parallel Prefix Computation", *J. Assoc. Comput. Mach.*, 27 (1980), 831-838.

[LV-88] G.M. Landau and U. Vishkin, "Fast parallel and serial approximate string matching", *J. of Algorithms*, 11 (1989), 157-169.

[MSV-86] Y. Maon, B. Schieber, and U. Vishkin, "Parallel ear decomposition search (EDS) and st-numbering in graphs", *Theoretical Computer Science*, 47 (1986), 277-298.

[RR-89] V. Ramachandran and J. H. Reif, "An optimal parallel algorithm for graph planarity", *Proc. 30th Symposium on the Foundations of Computer Science* (1989), 282-

287.

[S-88] Q. F. Stout, "Constant-time geometry on PRAMs", *ICPP 1988*, 104-107.

[SV-81] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model", *J. of Algorithms*, 2 (1981), 88-102.

[ScV-88] B. Schieber, and U. Vishkin, "On finding lowest common ancestors: simplification and parallelization", *SIAM J. Comput.*, 17,6 (1988), 1253-1262.

[ScV-88a] B. Schieber and U. Vishkin, "Finding all nearest neighbors for convex polygons in parallel: a new lower bound technique and a matching algorithm", UMIACS-TR-88-82, University of Maryland Inst. for Advanced Comp. Studies (1988). To appear in *Discrete Applied Math.*

[StV-84] L.J. Stockmeyer, and U. Vishkin, "Simulation of parallel random access machines by circuits", *SIAM J. Comput.*, 13 (1984), 409-422.

[SZ-89] D. Shasha and K. Zhang, "New Algorithms for the Editing Distance between Trees", *SPAA 1989*, 117-126. To appear in *Journal of Algorithms* as "Fast Algorithms for the Unit Cost Editing Distance Between Trees".

[Ta-75] R.E. Tarjan, "Efficiency of a good but not linear set union algorithm", J. Assoc. Comput. Mach., 22 (1975), 215-225.

[TV-85] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithms", *SIAM J. Comput.*, 14,4 (1985), 862-874.

[Va-75] L.G. Valiant, "Parallelism in comparison models", *SIAM J. of Computing*, 4 (1975), 348-355.

[Van-89] A. Van Gelder, "PRAM processor allocation: A hidden bottleneck in sublogarithmic algorithms" *IEEE Trans. on Computers*, 38,2 (1989), 289-292.

[Vi-85] U. Vishkin, "On efficient parallel strong orientation", *Information Processing Letters*, 20 (1985), 235-240.

[Vi-89] U. Vishkin, "Deterministic sampling for fast pattern matching", to be presented at *22 ACM Symp. on Theory of Computing* (1990). Also UMIACS-TR-2285, University of Maryland Inst. for Advanced Comp. Studies (1989).

[Y-82] A.C. Yao, "Space-time tradeoff for answering range queries", *Proc. 14 ACM Symp. on Theory of Computing* (1982), 128-136.

Number of children per node	Number of leaves per node	Level of the tree
$I_1^{(0)}(n)/I_1^{(1)}(n)=2$	$I_1^{(0)}(n)=n$	1
$I_1^{(1)}(n)/I_1^{(2)}(n)=2$	$I_1^{(1)}(n)=n/2$	2
$I_1^{(2)}(n)/I_1^{(3)}(n)=2$	$I_1^{(2)}(n)=n/4$	3
0	1	$*\ I_1(n)+1$ $= I_2(n)+1$ $= \log n + 1$

BT(2)

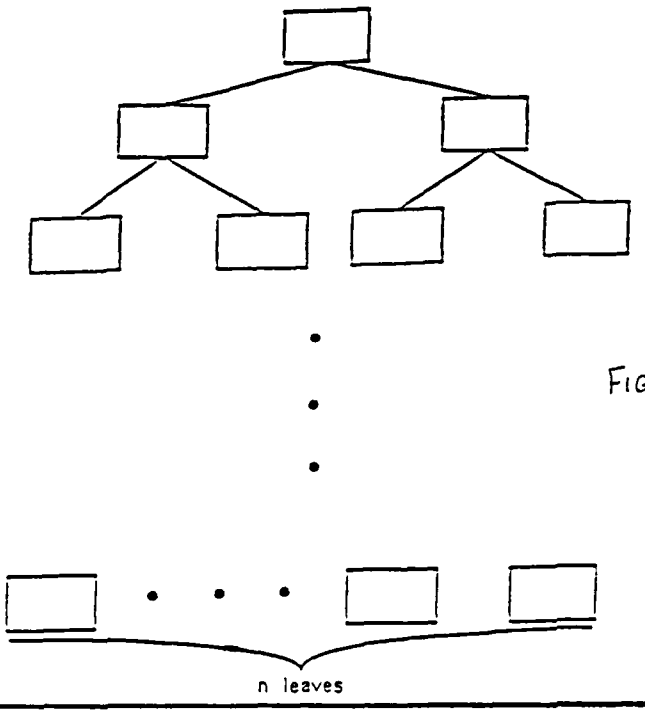


FIGURE 2·1

BT(m) – m'th recursive *-tree

$I_{m-1}^{(0)}(n)/I_{m-1}^{(1)}(n)$	$I_{m-1}^{(0)}(n)=n$	1
$I_{m-1}^{(1)}(n)/I_{m-1}^{(2)}(n)$	$I_{m-1}^{(1)}(n)$	2
$I_{m-1}^{(2)}(n)/I_{m-1}^{(3)}(n)$	$I_{m-1}^{(2)}(n)$	3
0	1	$*\ I_{m-1}^{(n)}+1$ $= I_m^{(n)}+1$

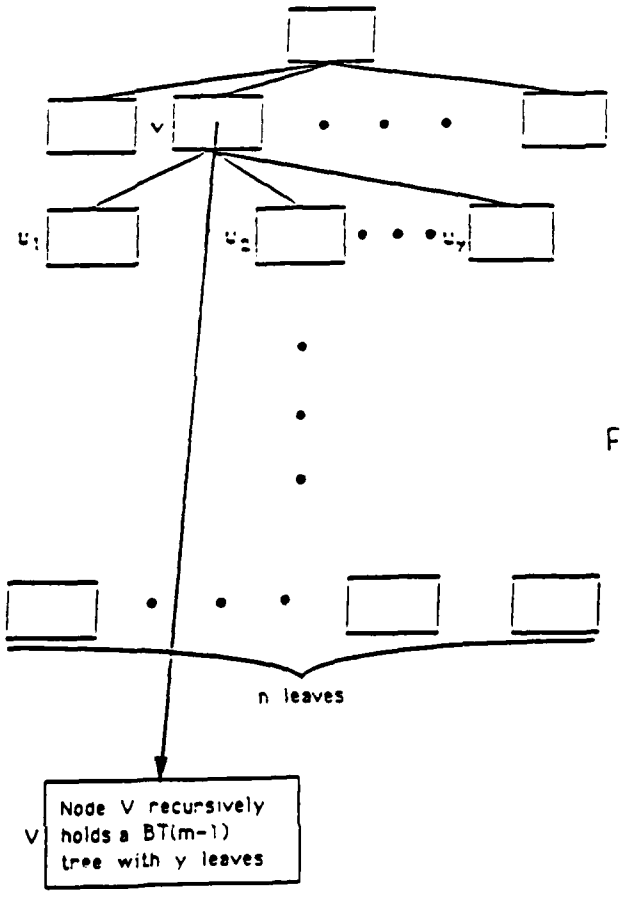
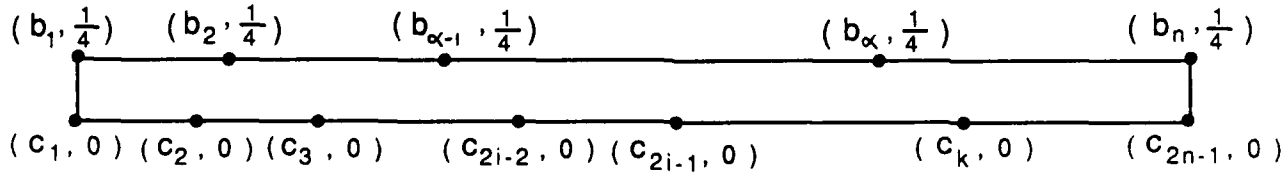


FIGURE 2·2



(where $c_1 = a_1$, $c_2 = \frac{a_1 + a_2}{2}$, • • • , $c_{2i-1} = a_i$, • • •)

Figure 5.1